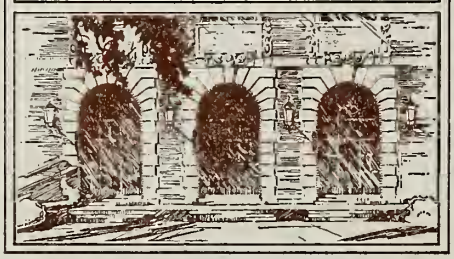
LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 619-626
cop. 2



370.84
Il6r
no.622 UIUCDCS-R-74-622
cop. 2
Math

Transportation of Higher-Level Language Programs: Exemplified by an ALGOL 68 Transportation Representation

by

Wilfred J. Hansen

January, 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE LIBRARY OF THE
MAR 4 1974
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

Digitized by the Internet Archive
in 2013

http://archive.org/details/transportationof622hans

UIUCDCS-R-74-622

Transportation of Higher-Level Language Programs: Exemplified by an ALGOL 68 Transportation Representation

by

Wilfred J. Hansen

January, 1974

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois

This research was supported by the Department of Computer Science.

Abstract

Problem: Modern higher level languages, for example, ALGOL 68, must be defined to allow many alternate token representations to accommodate the variety of input/output equipment available. Because of this variety, it may not be possible to transport programs from one hardware representation to another by any simple process of substitution.

Solution: As an example of the proposed approach, this paper defines a six-bit encoding for ALGOL 68 program texts. It is an encoded character prefix representation, so it can be decoded without lookahead. The encoding is defined in detail and a decoder for a specific hardware representation is given.

Acknowledgments

The concept of a syllable encoding was suggested to me by Charles Lindsey when I voiced an intention to produce a hardware representation proposal. (To the extent that his suggestion was intended to dissuade my intention, it was unsuccessful (2).) This paper would not have been completed without accepting (against inclination) C. A. R. Hoare's belief that "the quickest way to distinguish a good designer from a bad one is whether he prefers to avoid a problem rather than solving it." (5, p. 19) I hope I have avoided enough problems.

Table of Contents

page

The Problem: Transporting Program Texts..... 1

I. What criteria must a transportation representation satisfy?..... 3

II. What ALGOL 68 problems must a transportation representation face?.. 4

III. Structure of the proposed representation..... 5

IV. What special problems may face implementations?..... 6

IV.1. Token separation..... 6

IV.2. Multi-symbol representations..... 8

IV.3. Treatment of nonstandard graphics..... 8

IV.4. Encoders and decoders for implementations of variants..... 9

IV.5. Spaces and visible spaces..... 10

IV.6. What to convert to standard codes..... 10

IV.7. Puns..... 11

V. Conclusion..... 12

References..... 14

Appendix A

Specification of the ALGOL 68 Transportation Representation..... 15

A.1 Syllables and their functions..... 15

A.2 Representations of standard symbols..... 19

A.3 Summary of codes and standard symbols..... 21

Appendix B

Decoder from the Transportation Representation to a Hardware Representation..... 22

B.1 The hardware representation..... 22

B.2 Description of the decoder..... 22

B.3 The decoder..... 26

The Problem: Transporting Program Texts

One of the advantages originally claimed for higher level languages was machine-independence; the "same" program would run on many different machines. Experience has revealed that such languages are even more valuable as programming tools because they enhance communication between programmer, debugger, and modifier. Nonethless, there is still considerable need for machine independence and the transportation of programs between installations. For example, reference (4) defines a subset of FORTRAN that is accepted by most compilers; references (3) and (2) define ALGOL 68 representations that require only a reasonable, widely available set of graphics. (The transportation representation proposed below in no way depends on the hardware representation suggestions in (2). Those suggestions were made only to precisely define a vehicle for the decoder in Appendix B.)

In many cases, as J. McCarthy (7) has pointed out, the concrete syntax of a language is less critical than the abstract syntax - the inherent expressive features of the language. At least one recent language, Revised ALGOL 68 (1, hereinafter called the "Report"), has reflected this lesson; the defining document specifies the abstract syntax with great precision, but leaves many of the details of token representation to the discretion of implementors. This permits the freedom necessary to implement the language on a wide variety of input/output devices, but raises a new problem: token representations may be so incompatible that programs cannot be transported by any process of simple substitution.

This paper will first explore the dimensions of the problem and then describe a solution for ALGOL 68 in the form of a "transportation representation" text encoding for interchange of programs. Without this representation, program sharing among n implementations would require n^2 transliteration programs; with it, each implementation need have only an encoder and a

decoder for a total of 2n transliterators. (One can conceive of a compiler with numerous pragmat selected front ends to accept many hardware representations, but the multiplicity of the latter seems to doom the former.) A further advantage of the transportation representation will be the possibility of providing a decoder to translate to a publication language.

Simple substitution may not suffice to transport program texts for several reasons. Sometimes one symbol is used for several functions. For example, in ALGOL 68 the symbol " " must be transliterated to either entier or lwb depending on the context. More serious problems are posed by "stropping," the techniques used to distinguish boldface text from roman text. Some implementations demand apostrophes around boldface (whence the name stropping); others require backspacing and underlining; still others reduce clutter by specifying that boldface words are restricted and can never be used with a non-bold meaning. Many transformations between stropping conventions require complex analysis, including possibly tables of reserved words. A third problem occurs if the text contains lines longer than those accepted by the target compiler. Efforts to preserve original indentation are warranted because of its value in reading and modifying the text. Unfortunately, in contexts like bold words and strings it is not possible to freely intersperse spaces and new lines. Proper treatment of overlength lines is, in fact, a very context dependent problem.

I. What criteria must a transportation representation satisfy?

⁘ It must encode as much information from the original as possible. Where ALGOL 68 provides a choice between brief and bold forms, the programmer's choice should be recorded. When the encoding of '(b|c|d)' is decoded it should not become 'if b then c else d fi'. Original spacing should be recorded so some semblance can be reproduced, depending on the line length of the decoding media.

⁘ Text "close" to ALGOL 68 ought to be accepted as well as perfect programs. Programs should not be transported with syntax errors, but someone will probably want to do it. Moreover, this provision will make it possible to transport programs written in super-sets of ALGOL 68.

⁘ To be transportable to the maximum number of locations, the representation should be encoded in a small byte size. Six bits (64 unique values) can be represented conveniently on the transput media for almost all machines.

⁘ The representation design should emphasize simplicity of decoding. Encoders can be based on a compiler's existing token scanner.

⁘ The representation should not expand the size of the text by any significant factor. The proposed representation collapses multiple blanks to a two syllable code, so its output is significantly more compact than, say, cards. For magnetic tape, this factor would be less important.

II. What ALGOL 68 problems must a transportation representation face?

⁘ Represent two type faces, roman and bold, and two cases, upper and lower.

⁘ Contend with unusual graphics specified in the Report; consider '□', '$_{10}$', '°', not to mention the proposal in (2) for '≮'. Contend with graphics not mentioned in the Report but available at particular installations and used as "other monads" or "other string items."

⁘ Deal with string and pragment tokens which can contain diverse graphics and even non-graphics.

⁘ Solve the ambiguities inherent in multiple symbols for a single operator. For ease of decoding, '|' should be encoded not as just itself but as to whether it represents itself, lwb, or entier.

⁘ Consider the problems of implementations that allow, for instance, different representations of ':' in its contexts as, say, label-symbol and routine-symbol.

⁘ Devote some attention to the existence of national variants of ALGOL 68. Should the transportation representation provide for several alphabets? What should be done about national variants of the bold-tags, the tags defined in the standard prelude, and the letters in real and bits denotations and in format texts.

III. Structure of the proposed representation.

Three approaches to a transportation representation seem possible: (1) a standard representation in, perhaps, ASCII, (2) an encoded token level transliteration, (3) a syntax oriented representation.

The first would seem a chimera since it would be difficult to get general agreement and because it could not encode all the token structure and unusual graphics that might be available in some implementations. It would have the advantage of being both human readable and machine readable.

The third is very attractive from a number of viewpoints. Only the abstract syntax would be recorded so a decoder could easily choose whatever label-symbol or routine-symbol was applicable. Unfortunately, a syntactic representation takes more storage (see the Appendices in (6)), is complex to encode and decode, and implies some standard, agreed upon grammar.

Based on these considerations, I chose the second option, a token level prefix code that can be decoded without look-ahead. The encoded form of the text is a sequence of operator, letter, and digit syllables, represented as six bit quantities. These syllables are specified in detail in Appendix A.

Syllables in the code can change the case and the type face of succeeding letters. Special ALGOL 68 symbols and operators defined in the standard prelude are represented by a prefix syllable (#7) followed by a syllable to select the symbol (quote-symbol is thus #7, #2; '↑' as a left-shift-operator is #7, #35). The decoder is simplified by assignment of different codes to each of the possible uses of '↑', '~', and '['. Provision is made for 6 "national letters," 64 other nonstandard letters, and 4160 "other monads." To reduce the size of encoded text, multiple

blanks are represented with just two syllables, and there is provision to assign any syllable to invoke a sequence of syllables.

Unlike special character symbols, there is no implied list of bold-tags or standard prelude tags; these are simply spelled out using appropriate type face changes. Consequently, superset languages may use this transportation representation even if they have additional standard bold-tags.

IV. What special problems may face implementations?

IV.1. Token separation

Trivial but annoying problems are posed to encoders and decoders by the need to determine transitions between roman tags, bold tags, and numbers. These problems are made a little more difficult if the hardware representation of the implementation uses reserved word stropping, but they are hardly insurmountable. To avoid expansion of the text, no codes are transmitted to specify token types. A decoder must keep track of the current token type to provide for proper separation. In fact, this requirement is the reason for inclusion in the decoder below of the non-trivial routine, CHANGE_STATE.

Encoders in general may omit face shifts if tags are separated by spaces or special characters. Thus only one face shift at the beginning would be needed for either

```
abs long 1 + abs 0
```

or

```
x + sin(y)/z2 + 7.
```

The spaces separate the bold tags and the special characters separate the roman tags. The encoder may need to insert separation in several cases:

a) Stropped bold tags separated only by strop characters: 'ref''real'.

The encoder may insert spaces for strop characters, but preferably it will only mark the separation by inserting a bold face shift.

b) Stropped bold tag followed by a number.

Again, a face shift may be inserted.

c) Number followed by a tag.

This is perfectly legal and no separation would be needed.

Some implementations will mark spaces in tags in some way. But, because tags and non-tag letters may appear in format, it is not always trivial to determine when a space is within a tag. For this reason, encoders must specially mark such spaces (with the token space code, #2). Encoders for reserved word hardware representations can best be built as an adjunct of the compiler and use the compiler's routines for distinguishing bold and roman.

Decoders can easily distinguish bold from roman, but must properly strop the bold. The three cases discussed for encoders must be considered for the decoder, too. Where a bold tag is separated from a succeeding bold tag or number by no more than a face shift code, the decoder must insert stropping characters or spaces to separate them. If prefix stropping is used, a bold tag immediately after a number must be detected so the stropping can be inserted.

To avoid confusion of roman and bold, reserved word implementations must check every roman tag for equality to a bold tag and modify the roman in some way--perhaps by the addition of an "x".

IV.2. Multi-symbol representations

A few implementations may choose distinct representations for distinct symbols that are assigned the same representation by the Report (e.g., ':' suffices for four distinct symbols in the Report but some implementation may use another representation for one of the four--perhaps

isdefinedtoimplyresumptionatthispoint

for label-symbol). The encoder for such an implementation will map the distinct representations into the single standard code provided by the transportation representation (#26 represents ':'). The decoder must solve the more context dependent problem of distinguishing among the various meanings of the standard code. This decision to represent only the basic graphics in the transportation representation means that the majority of encoders will be simplified at the expense of a small number of decoders.

IV.3. Treatment of nonstandard graphics.

The proposed transportation representation provides for both 'national letter' and 'other monad' special characters. These characters may appear in five different contexts: letters appear in tags and bold tags, monads in TAO's, and both appear in strings and pragments. An encoder for an implementation that allows special characters must assign one of the appropriate codes to each character and output that code for each occurrence

of the character. Accompanying the encoded text should be documentation specifying the characters the encoder has associated with each special code.

Decoders must react appropriately to each type of special character in each of the five contexts. Default actions for each case are specified in Appendix B. In earlier versions the decoder accepted run-time input for (1) specification of transliterations for all special marks; (2) specification of notes--where the special character occurs, a pointer appears under the transliteration and the text of the note appears in the righthand margin. Another option a decoder might provide would be a specification of whether a TAO containing a special character should be transliterated. The user might not want this to occur if he was able to specify a monad to replace the special character in TAO's.

IV.4. Encoders and decoders for implementations of variants.

Like the Report itself, this proposal is slanted toward the English alphabet and English identifiers in the standard prelude. Considerable thought was devoted to allowing a kind of alphabet shift--similar to a case shift--to specify that the text was constructed from characters in some other alphabet. (Possibly even providing a representation for letter-aleph!) With this scheme, however, every decoder would have had large tables of transliterations and considerable code that would in practice have been rarely used. Instead, this extra effort should be expended only by implementers for variant languages. They will transliterate into English to transport programs with the representation proposed here.

Several classes of token must be translated to English: standard bold tags, tags in the standard prelude, and letters in denotations and format texts. Other letters can be strictly transliterated in such a way that when the text is decoded back into its original language, it will be the same. The text will probably be meaningless English, but it could be compiled with an English speaking compiler. If used in this way, incidentally, there is no reason why the transportation representation cannot serve equally well to transport between two variant implementations.

IV.5. Spaces and visible spaces

Some ALGOL 68 implementations will not provide a representation for visible-space. They must still encode spaces within strings to the standard representation for visible-space. Moreover, on decoding they must ignore ordinary spaces in strings and convert visible-spaces to spaces.

IV.6. What to convert to standard codes

Certain symbols are represented by both standard graphics and bold tags; the question arises as to when to encode a bold tag as the corresponding standard symbol. For example, nil in a program may be encoded as either nil or standard code #62. If the latter option were chosen, when it was decoded it might appear as '°', depending on the character set available.

In general it is preferable to encode a bold tag as a bold tag so the decoded text is as close as possible to the original. Appendix A.2 reflects this by enclosing bold tag representations in parentheses in the fourth column. This signifies that any encoder I write will not output the standard code on the left (using the bold tag instead), but the decoder will respond to the standard code by producing the given bold tag.

A few graphics are well defined and widely available but also have bold alternatives. These I would encode as the standard code: NE, LT, LE, GE, GT, EQ, NOT, AND, OR, OF, CO. In addition, I would encode an 'e' in a real denotation as the code for times-ten-to-the-power. To keep the encoder simple I would convert style-i-sub(or bus)-symbol to open(or close)-symbol rather than to brief-sub(or bus)-symbol.

IV.7 Puns

A 'pun' occurs when two different sequences of codes map into the same ALGOL 68 text. For example, the decoder in Appendix B can generate puns six ways:

1) Appending "x" to tags that would otherwise be mistaken for bold reserved words. (The tags "real" and "realx" both map into "realx".) The generation of the "x" is an attempt to prevent the more serious pun of converting an input roman tag (which must have been stropped in the original text) into a bold tag.

2) Transliteration of special letters and monads. ("Zy" and code 32 both map into "Zy". The bold tag QO and the first "other monad" both map to "QO'".)

3) Transliteration for illegal standard code operands. (ERR18 and standard code 18 both map to "ERR18'".)

4) Transliterations for legal standard codes. (ELEM and standard code 34 (window) both map to "ELEM'".)

5) Representation of two different nomads--asterisk and times--with "*".

6) Representation of standard codes with diphthongs. ("+*" (as two standard codes) and standard code 41 (plus-i-times) both map to "+*".)

Puns will seldom occur in practice, but for this reason they are all the more dangerous--their possible existence may be forgotten. When they do occur, they will usually generate syntax errors during a subsequent compilation. If not, the worst has occurred, and lengthy frustration may ensue. To guard against this, the decoder must maintain a symbol table and take steps to correct all possible puns.

V. Conclusion

It should be noted that this proposal is only a small step toward machine independence. The far more crucial (and difficult) problems of arithmetic precision and bits- and bytes-widths still remain. Perhaps these can be remedied by some informal agreement that so many long's specify at least so much precision. The compiler would then give a warning message if it could not satisfy a precision request. It has been suggested that mode definitions can provide machine independence; however, denotations must have their length specified, and there is no coercion of length.

Implementors' efforts to provide encoders and decoders for the transportation representation will permit wide interchange of ALGOL 68 programs. This is one more step toward machine independence and freedom from the dictates of computer manufacturers.

References

(1) van Wijngaarden, A., et al., Almost the Revised Report on the Algorithmic Language ALGOL 68, private communication (Sept., 1973). This version is slightly more recent than the version accepted at Los Angeles and includes most of the corrections agreed on there.

(2) Hansen, W. J., A Revised ALGOL 68 Hardware Representation for ISO-code and EBCDIC, submitted to ALGOL Bulletin (Nov., 1969).

(3) Lindsey, C. H., "An ISO-Code Representation for ALGOL 68," ALGOL Bulletin 31 (March, 1970), pp. 37-60.

(4) Hall, A. D. Jr., "A 'Portable' FORTRAN IV Subset," Bell Telephone Laboratories, Murray Hill, New Jersey, Sept., 1969, unpublished; cited in Ryder, B. G., "The FORTRAN Verifier: User's Guide," Computing Science Tech. Rep. #12, Bell Telephone Laboratories, Murray Hill, New Jersey, March, 1973.

(5) Hoare, C. A. R., "Hints on Programming Language Design." Draft manuscript distributed at SIGPLAN/SIGACT Symposium on Principles of Programming Languages, Boston (Oct., 1973).

(6) Hansen, W. J., "Creation of Hierarchic Text with a Computer Display," Ph.D. Thesis, Stanford University, June, 1971.

(7) McCarthy, John, "Towards a Mathematical Science of Computation," in Information Processing, 1962 (C. M. Popplewell, Ed.), North-Holland Publ. Co., Amsterdam, 1963, pp. 21-28.

Appendix A. Specification of the ALGOL 68 Transportation Representation

An ALGOL 68 program is represented in the Transportation Representation as a sequence of six-bit syllables. Some syllables represent characters directly (#25 represents the character '9'), some sequences represent characters (#7, #33 is '&'), and some syllables only affect the state of the decoder (#11 switches to boldface). To facilitate debugging, the meaning of each code has been assigned with reference to the ISO-codes: code i is related to the ISO character at position i+32 (e.g., #33 is 'A' and ISO character 65 is 'A').

The definitions of the syllables are given in section A.1. Section A.2 specifies the representation of standard symbols and A.3 summarizes both the definitions and the standard symbols.

A.1 Syllables and their functions

a. Syllables that represent standard symbols directly:

#0	space	#1	|	#8	(
#9	)	#12	,	#26	:
#27	;	#29	:=		

b. Syllables that represent digits and letters:

#16	0	#33	A
...		...	
#25	9	#58	Z

c. Syllables that represent unassigned letters:

#32, #59, #60, #61, #62, #63

For each of these, a decoder may substitute letters not used for codes #33 through #58 or some carefully chosen string.

d. Syllables with assigned functions:

(Parenthesized terms indicate that succeeding input codes serve as arguments to the function syllable.)

#14 - consecutive spaces (number of spaces)

The argument syllable is interpreted as a number of spaces to be inserted in the output text.

#2 - token space

This syllable encodes a space occurring within a tag or number. It must be the first syllable in any string of typographical display features (codes #14, #15, #5) in a tag or number.

#15 - newline (number of spaces)

A new line of output text is begun and started with the indicated number of spaces.

#5 - new page (number of syllables)

A new page of output is begun and its first line initially contains the specified number of spaces. (Decoders may treat this function as identical to #15 (new line).)

#28 - upper case

Succeeding output letters are upper case. Case does not affect the graphic representation of digits and special characters. Decoders may ignore case shifts if the output representation has only one case or uses case for stropping. To preserve the appearance of the text, encoders for representations that strop with case should shift to bold case and the strop case simultaneously; but case shift is never to be interpreted by decoders as an indication of stropping.

#30 - lower case

Shift output letters to lower case. See the remarks for #28 (upper case).

#11 - boldface

Succeeding letters and any digits following letters are output as bold face (and stropped as necessary). Type face has no effect on special

characters and digits not following letters. The boldface shift may be omitted between two bold tags if they are separated by spaces or special characters. Bold shift may be inserted to separate two bold tags which in the original were separated only by stropping characters. See the discussion in Section IV.1.

#13 - roman face

Succeeding characters are to be roman. This code may be omitted between two roman tags if they are separated by special characters. See the discussion in Section IV.1.

#7 - standard symbol (symbol code)

The next code byte selects a standard symbol (according to tables A.2 and A.3). For example, a quote-symbol is #7, #2, quote-image-symbol is #7, #7, and a visible-space is #7, #63. #7 (standard symbol) may also select any of the codes noted in section A.1.a which represent standard symbols directly (thus #7, #8 could be used for open-symbol instead of #8 alone).

The standard symbols represent <u>functions</u> rather than symbols in Chapter 9 of the Report. Thus there are, for example, three standard symbols corresponding to the Report's tilde-symbol: #52 (tilde-symbol), #47 (skip-operator), and #43 (negation-operator). Moreover certain operators are represented, though they appear only in Chapter 10: for example #45 (modulo-operator). Where a symbol only functions as a single operator, only one standard symbol is provided: e.g., #36 (down-symbol) for the shift-right-operator.

#3 - pragment mark (pragment delimiter, pragment mark, pragment, pragment mark)

Because there are many pragment delimiters, the job of the decoder is facilitated if there is a unique code to signal the bounds of a

pragment. The original pragment delimiter must be preserved, so it is encoded between two pragment marks and they are followed by the pragment and a third pragment mark. There is a net reduction of text if the delimiter is comment or pragmat, and in other cases the encoded text can be shortened with judicious use of #10 (define).

#31 - other monad (monad number)

Various character sets support various other monads. If an encoder is written for a character set with N other monads, they will be assigned monad numbers one to N. A decoder can represent these with other monads available in the target character set or with a transliterated bold tag. Other monads may also appear in strings and pragments; here they represent some symbol an explanation of which should accompany the encoded text when it is transported.

#4 - 4096 monads (first digit of monad number, second digit of monad number)

Some installations will have more than 64 other monads. The excess may be encoded with this code, where the digits of the monad number are interpreted in base 64. The remarks with #31 (other monad) apply.

#6 - other letter (letter number)

If the six codes allocated for other letters are not enough, 64 more letters can be represented with this code. The decoder may replace each such letter with any suitable transliteration.

#10 - define (code defined, definition length, definition string)

To further shorten the encoded text, single codes may be assigned a string of codes as their meaning. The codes listed in sections A.1.a and A.1.c above are readily available for reassignment, but other codes can be used. If a code is assigned a definition string of length zero, its original interpretation is restored. For example, a comment delimited by

hash marks would be encoded as "#3, #7, #3, #3, text of comment, #3." This could be shortened if the unassigned code #32 were defined to have the value #3, #7, #3, #3. The definition would be established by the sequence #10, #32, #4, #3, #7, #3, #3.

A.2 Representations of standard symbols

code[1]	symbol	representations reference	representations hardware[2]	notes
14	point symbol	.	.	
37	times ten to the power symbol	ıo	e	
60	times ten to the power alter.	\	(e)	
2	quote symbol	"	"	
7	quote image symbol	""	""	
0	space symbol	space	space	7
63	space alternate	.	(space)	
54	or symbol	∨	or	4
6	and symbol	∧	&	4
33	ampersand symbol	&	(am)	4
58	differs from symbol	≠	/=	4
28	is less than symbol	<	<	5 6
44	is at most symbol	≤	<=	4
39	is at least symbol	≥	>=	4
30	is greater than symbol	>	>	5 6
15	divided by symbol	/	/	5 6
5	over symbol	÷	%	4
45	modulo operator	÷×	%*	8
25	percent symbol	%	(pc)	4
34	window symbol	▯	(elem)	4
22	floor symbol	⌊	(fl)	4
21	entier operator	⌊	(entier)	8
20	lower bound operator	⌊	(lwb)	8
23	ceiling symbol	⌈	(upb)	4
41	plus i times symbol	⊥	+*	4
46	not symbol	¬	¬ or ~	4
52	tilde symbol	~	(tl)	4
47	skip operator	~	(skip)	8
43	negation operator	~	(ng)	8
36	down symbol	↓	(shr)	4
53	up symbol	↑	(up)	4
50	raised to operator	↑	(**)	8

1 This code must follow the function code #7 (standard symbol) to invoke the symbol or operator shown in the second column.

2 See Hardware Representation (2). Parentheses around a graphic indicate the encoder (as discussed in section IV.6) will never output the corresponding code, but in response to the code the decoder in Appendix B will produce the graphic.

More notes on next page.

code	symbol	representations reference	representations hardware	notes	
35	shift left operator	↑	(shl)		8
48	power operator	**	**		8
11	plus symbol	+	+	4	
13	minus symbol	-	-	4	
16	equals symbol	=	=	5 6	
10	times symbol	×	*	5 6	
51	asterisk symbol	*	(*)	5 6	
17	assigns to symbol	=:	=:	6	
29	becomes symbol	:=	:=	6	7
12	comma symbol	,	,		7
27	semicolon symbol	;	;		7
26	colon symbol	:	:		7
8	open symbol	(	(		7
9	close symbol	)	)		7
1	stick symbol	|	?		7
55	again symbol	|:	?:		
59	brief sub symbol	[	(()		
61	brief bus symbol	]	())		
32	at symbol	@	@		
57	is symbol	:=:	:=:		
56	is not symbol	:≠:	:/=:		
62	nil symbol	°	(nil)		
38	of symbol	≮	-<		
4	formatter symbol	$	$		
31	brief comment symbol	¢	co	3	
3	style ii comment symbol	#	#		

---

3 In ISO/ASCII, co maps to co and {...} maps to ¢ ... ¢.

4 This standard symbol is a monad defined in the Report.

5 Nomad defined in the Report.

6 Because in a TAO this code may follow a transliterated monad, the following transliterations are sometimes used

code	28	30	15	16	10	51	17	29
symbol	<	>	/	=	×	*	=:	:=
transliteration	LT	GT	DV	EQ	TM	ST	TO	AB

7 The code at the left is a function code that will invoke this standard symbol directly. (E.g., #29 alone may be used instead of #7, #29 to encode the becomes-symbol.)

8 This symbol is not a monad and is so treated by the decoder in Appendix B.

A.3 Summary of codes and standard symbols

code	ISO	function	std. symbol	code	ISO	function	std. symbol
0	space	space	space	32	@		@
1	!	|	|	33	A	A	&
2	"	token space	"	34	B	B	▯
3	#	pragment	#	35	C	C	↑(shl)
4	$	4096 monads	$	36	D	D	↓
5	%	new page	÷	37	E	E	⌽
6	&	other letter	^	38	F	F	⊲
7	'	std. symbol	""	39	G	G	⩾
8	(	(	(	40	H	H	
9	)	)	)	41	I	I	⊥
10	*	define	×	42	J	J	
11	+	bold	+	43	K	K	~(neg)
12	,	,	,	44	L	L	⩽
13	-	roman	-	45	M	M	÷×
14	.	spaces	.	46	N	N	¬
15	/	newline	/	47	O	O	~(skip)
16	0	0	=	48	P	P	**
17	1	1	=:	49	Q	Q	
18	2	2		50	R	R	↑(pow)
19	3	3		51	S	S	*
20	4	4	⌊(lwb)	52	T	T	~
21	5	5	⌊(entier)	53	U	U	↑
22	6	6	⌊	54	V	V	⌄
23	7	7	⌈	55	W	W	|:
24	8	8		56	X	X	:≠:
25	9	9	%	57	Y	Y	:=:
26	:	:	:	58	Z	Z	≠
27	;	;	;	59	[		[
28	<	upper case	<	60	\		\
29	=	:=	:=	61	]		]
30	>	lower case	>	62	^		°
31	?	other monad	∉	63	_		÷

code — value of six-bit code byte to invoke this function and standard symbol.

ISO — graphic in position code+32 in the ISO code.

function — the action or graphic selected by this code as a function.

std.symbol — standard symbol selected if this code appears as an operand to code 7.

Appendix B. Decoder from the Transportation Representation to a Hardware Representation

B.1 The hardware representation

The decoder is written in--and produces text in--the ISO/ASCII/EBCDIC hardware representation defined in (2). The principle differences from the reference language are '-<' for of, '?' for then and else, and in or out for '|' meaning in or out.

With three exceptions, bold tags defined in the report are reserved as recommended in (2). User defined bold tags and i, im, and re must be marked by a final apostrophe. Case is optional for all tokens, but in writing the program below I exercised this option to make bold tags upper case. (The program output depends on the case specified by its input.)

Tags and numbers may contain typographical display features (R9.4d), but each string of them in a tag must contain at least one underline ("_").

As a concession to this decoder, the compiler allows the following diphthongs as monads: +*, >=, <=, and /=. (+* is the representation of the plus-i-times-symbol.) This provision allows operators like >=< and /=/, but no ambiguity arises. Indeed, no ambiguity would arise if any number of nomads were allowed in operators, instead of at most two.

B.2 Description of the decoder

The input is on the file 'CODE_IN', a stream of six bits code syllables as defined in Appendix A. The output is a "reconstituted" ALGOL 68 program on the file 'RA68_OUT'; a listing is also produced on 'STAND_OUT'.

In general, the main routine of the decoder reads a code from the input and uses its value to select and execute an element of the 'row-proc-void' CODE_FUNCTION. The element is usually one of the procedures whose name begins with 'PUT_'. These in turn append characters to the output stream with the operator '+/:='.

Because typographical display features are illegal in bold tags and because string continuations may not be indented, it is necessary for the decoder to keep track of what type of token it is processing. At each point that might be the start of a token, CHANGE_STATE is called. If necessary, it terminates the old token and starts the next; it may introduce blanks or stropping characters. These functions are controlled by an automaton with nine states and nine inputs as shown in Figure B.1. Before exiting, CHANGE_STATE usually calls SET_BREAK to store the location of the beginning of the new token. If the old token extends beyond the maximum line length, SET_BREAK calls BREAK_LINE to interrupt the line. In most cases, the entire old token is moved to the new line and the new line is indented seven spaces more than the last indentation specified with a newline code.

Certain inputs are ignored:

- newlines in strings,
- all spaces in a tag following a 'token space' code,
- spaces following a point where a line was broken (if the break was due to spaces on the previous line).

Errors are treated by just counting them. In a production decoder, they would be indicated appropriately on the STAND_OUT listing. I wrote the code to do this, but it added bulk without light and was discarded from the current version. The following errors are detected:

A 'token space' code used outside a tag or number.

A 'token space' code in a tag but not followed by a continuation of the tag.

A 'token space' code in a number, but not followed by a continuation of the number.

An illegal operand for 'standard code'.

A TAO that has too many nomads.

A 'visible space' occurring outside a string.

An appropriate action is taken in each case, and processing continues.

Input Symbol	SPACE_IN	TOKSP_IN	STD_IN	TAO_IN	PMENT_IN	ROM_IN	BOLD_IN	DIG_IN	QUOT_IN
Default New State	OUT_ST	OUT_ST	OUT_ST	TAO_ST	PMENT_ST	TAG_ST	BOLD_ST	NUM_ST	STR_ST
Default Tag Start	TAGSTART = TRUE					TAGSTART = FALSE			
Current State: OUT_ST	/?	ERR*	SS=F /?	SS=F /?	SS=F /?	SS=F /?	SS=F /?	SS=F /?	*SS=T /?
TAO_ST	TC=0	ERR* TC=0	TC=0 /?	TC=0 /?	TC=0	TC=0	TC=0	TC=0	*SS=T TC=0
BOLD_ST	TC=0 "," /?	ERR* "," /? TC=0	TC=0 "," /?	TC=0 "," /?	TC=0 "," /?	TC=0 "," /?	TC=0 "," /?		*SS=T "," /? TC=0
TAG_ST	"x" US=T /?	TAGSP_ST US=F /? SS=T	"x" US=T /?	"x" US=T /?	"x" US=T /?		"x" US=T "." /?		"x" *US=SS=T /?
TAGSP_ST	TAGSP_ST	TAGSP_ST /?	US=T SS=F /? ERR	US=T SS=F /? ERR	US=T SS=F /? ERR	SS=F " "	US=T SS=F /? ERR	TAG_ST SS=F " "	*US=SS=T ERR /?
NUM_ST	/?	NUMSP_ST " " /?	/?	/?	/?	"R"? " " /?	" " /?		*SS=T /?
NUMSP_ST	NUMSP_ST	NUMSP_ST /?	ERR /?	ERR /?	ERR /?	"R"? NOT"R"?	ERR /?	/?	*SS=T ERR /?
PMENT_ST	PMENT_ST /?	PMENT_ST ERR* /?	PMENT_ST /?	PMENT_ST /?	OUT_ST /?	PMENT_ST /?	PMENT_ST /?	PMENT_ST /?	PMENT_ST TS=T *SS=F /?
STR_ST	STR_ST TS=F	STR_ST TS=F ERR*	STR_ST TS=F	STR_ST TS=F	STR_ST TS=F	STR_ST TS=F	STR_ST TS=F	STR_ST TS=F	OUT_ST TS=T *SS=F

Figure B.1 Automaton to control token separation and line breaks (key on next page)

State name

A state name as an operation (TAGSP_ST, TAG_ST, NUMSP_ST, NUM_ST, PMENT_ST, STR_ST) signifies that the new state will be the named state rather than the default new state given atop the column.

Flag setting

A flag is set TRUE (or FALSE) with 'flag=T' (or 'flag=F'). Two flags may be set at once as in 'US=SS=T'. (SS* indicates an assignment in PUT_STD.) The flags are

TS - (TAGSTART) Default settings for this flag are indicated atop the columns. When a letter or digit is encountered, CHANGE_STATE is only called if TAGSTART is TRUE. It is also set TRUE by a typeface change code (BOLD or ROMAN).

SS - (SKIPPING_SPACES) When this flag is TRUE, spaces are ignored (but visible spaces are still converted to blanks and output). The flag is also set TRUE by BREAKLINE if a break occurs while in OUT_ST.

US - (UNSPACED) Used to control the operation "x" below.

Token separator output

Under certain conditions extra characters are output to separate tokens which would otherwise run together:

"x" - If preceeding tag is reserved and UNSPACED is TRUE, output an "x". (For example, this converts the tag REAL to REALX.)

"␣" - If the output line does not have exactly zero spaces left, output a blank.

"'" - If the preceeding bold tag is not reserved, output a "'".

"'␣" - If no apostrophe is output by a "'", then do "␣".

"_" - Put an underbar in a tag for a token space.

Miscellaneous

/? - Call SET_BREAK. Breaks line if previous token extends over end of line. Notes where token begins.

ERR - Call ERROR with a suitable message. (ERR* indicates the error call in PUT_TOKSP.)

"R"? - If CURR_CODE is an "R" then next state will be NUM_ST: ("␣" and /? are ignored).

NOT "R"? - If CURR_CODE is not "R" then call ERROR and SET_BREAK.

TC=0 - Set TAO_CNT to zero, to indicate that any subsequent nomad is the start of a TAO.

Notes for Figure B.1 Operations Used in Automaton

B.3 The Decoder

```
BEGIN COMMENT Decoder for Transportation Representation COMMENT

      #The perm_code_function array is the permanent list of
       procedures to be executed for each of the 64 input codes.
       To provide for definitions, the permanent array is copied
       into the code-function array, which is modified when a
       definition occurs.  The main program will be

             WHILE decoding DO code-function(getch) OD

       where getch gets the next code from the input file, code_in.#

(0:63)  PROC VOID perm_code_function, code_function;

FOR i FROM 16 TO 25 DO

       perm_code_function (i) := put_dig OD;

FOR i FROM 32 TO 63 DO

       perm_code_function (i) := VOID:  put_let (curr_code) OD;

perm_code_function (0:15) := (0:15) PROC VOID

       (#0 space #          VOID:  put_spaces(1),
       #1 '|' #             VOID:  put_std(1),
       #2 token space #     VOID:  put_toksp,
       #3 pragment #        VOID:  put_pragment,
       #4 4096 monads #     VOID:  put_bold_monad
              ("Q"+whole(64*getch+getch+64, 0)),
       #5 new page #        VOID:  put_line(getch),
       #6 other letters #   VOID:  put_let(getch+64),
       #7 std. symbol #     VOID:  put_std(getch),
       #8 '(' #             VOID:  put_std(8),
       #9 ')' #             VOID:  put_std(9),
       #10 define #         VOID:  set_define,
       #11 boldface #       VOID:  (curr_face := bold_face; tagstart := TRUE),
       #12 ',' #            VOID:  put_std(12),
       #13 romanface #      VOID:  (curr_face := roman_face; tagstart := TRUE),
       #14 spaces #         VOID:  put_spaces(getch),
       #15 newline #        VOID:  put_line(getch));
```

```
perm_code_function (26:31) := (26:31) PROC VOID

        (#26 ':' #          VOID:  put_std(26),
        #27 ';' #           VOID:  put_std(27),
        #28 upper case #    VOID:  curr_case := upper_case,
        #29 ':=' #          VOID:  put_std(29),
        #30 lower case #    VOID:  curr_case := lower_case,
        #31 other monads #  VOID:  put_bold_monad
                  ("Q" + whole(getch, 0)));

code_function := perm_code_function;


        # Line breaks and token delimitation are controlled by a finite
        state automaton which is activated at places of possible start
        of token.  The states and input symbols have these mnemonics:#

INT out_st = 1,  tao_st = 2,  bold_st = 3,  tag_st = 4,
        tagsp_st = 5,  num_st = 6,  numsp_st = 7,
        pment_st = 8,  str_st = 9;

INT space_in = 1,  toksp_in = 2,  std_in = 3,
        tao_in = 4,  pment_in = 5,  rom_in = 6,  bold_in = 7,
        dig_in = 8,  quot_in = 9;

        # These flags also maintain state information #

BOOL unspaced := TRUE, # TRUE except during that part of a tag that
        follows a space in the tag (which is replaced by an underbar) #

     tagstart := TRUE, # set TRUE whenever the next digit or letter
        could start a token #

     skipping_spaces := false; # TRUE after underbar inserted in tag
        or sometimes after breaking a line.  Causes spaces to be
        ignored. #

INT curr_state := out_st;  # remember what state we are in #

PROC change_state = (INT input) VOID:

     (INT next_state := (input IN out_st, out_st, out_st,
        tao_st, pment_st, tag_st, bold_st, num_st, str_st);

        tagstart := input <= pment_in;

        CASE curr_state IN

#1 out_st #     IF input /= toksp_in THEN set_break;

                    IF input /= space_in THEN

                        skipping_spaces := FALSE FI FI,
```

```
#2 tao_st #    (tao_cnt := 0;
          IF input = std_in OR input = tao_in
               THEN set_break FI),
#3 bold_st #   IF input /= dig_in THEN
           IF NOT is reserved (last_tok_out) THEN +/:= "'"
           ELIF (input = bold_in OR input = rom_in)
                     & NOT line_has_left(0)  THEN +/:= " " FI;
           set_break;    tao_cnt := 0   FI,
#4 tag_st #
           IF input /= rom_in AND input /= dig_in THEN
                IF input = toksp_in THEN
                     next_state := tagsp_st;
                     skipping_spaces := TRUE;
                     unspaced := FALSE
                ELSE IF unspaced & is_reserved(last_tok_out)
                          THEN +/:= "x" FI;
                     unspaced := TRUE;
                     IF input = bold_in
                               & NOT line_has_left(0)
                          THEN +/:= " " FI
                FI;
                set_break FI,
#5 tagsp_st #
           IF input = space_in THEN next_state := tagsp_st
           ELIF input = rom_in OR input = dig_in THEN
                skipping_spaces := FALSE;
```

```
                next_state := tag_st;
                +/:= "_"
            ELIF input = toksp_in THEN
                next_state := tagsp_st;
                set_break
            ELSE error ("Tag space not followed by more tag.");
                unspaced := TRUE;    skipping-spaces := FALSE;
                set_break   FI,
#6 num_st #     IF input /= dig_in THEN
                IF input = rom_in & curr_code = 50 # an "r" # THEN
                    next_state := num_st
                ELSE IF (input = rom_in OR input = bold_in
                        OR (input = toksp_in
                            ? next_state := numsp_st;   TRUE
                            ? FALSE))
                       & NOT line_has_left(0)
                    THEN +/:= " " FI;
                    set_break
                FI FI ,
#7 numsp_st #
            IF input = space_in THEN
                next_state := numsp_st
            ELIF input = toksp_in THEN
                next_state := numsp_st; set_break
            ELIF input = dig_in THEN set_break
            ELIF input = rom_in & curr_code = 50 # an "r" #
                THEN next_state := num_st
```

```
            ELSE error ("Space in number not followed by number.");
                 set_break FI,
#8 pment_st #    (set_break;
            next_state := (input = pment_in ? out_st ? pment_st);
            IF input = quot_in THEN tagstart := TRUE FI),
#9 str_st #
            IF input = quot_in THEN
                 tagstart := TRUE;    next_state := out_st
            ELSE tagstart := FALSE;   next_state := str_st FI
     ESAC;
     curr_state := next_state); # end of change_state #

     # The following 'put_' routines are called on in response to
     specific codes from the input stream #
PROC put_dig = VOID:
     (IF tagstart THEN change_state (dig_in) FI;
     +/:=  "0123456789"(curr_code - 16 # a "0" # + 1));
INT roman_face := rom_in, bold_face := bold_in,
     upper_case = 2, lower_case = 1;
INT curr_face := roman_face, curr_case := lower_case;
PROC put_let = (INT let) VOID:
     (IF tagstart OR curr_state = num_st
            THEN change_state (curr_face) FI;
     +/:= IF let > 63 THEN "Z" + whole(let,0)
            ELIF let >= 59 THEN "Z" + "aeiou"(let-58)
            ELIF let = 32 THEN "Zy"
            ELIF curr_case = upper_case THEN
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"(let-32)
            ELSE "abcdefghijklmnopqrstuvwxyz"(let-32)  FI);
```

```
PROC put_toksp = VOID:
     (IF curr_state <= bold_st OR curr_state >= pment_st THEN
            error ("Token space outside tag or number.") FI;
     change_state (toksp_in));
PROC put_spaces = (INT nsp) VOID:
     (change_state(space_in);
     IF NOT skipping_spaces THEN +/:= nsp*" " FI);
PROC put_line = (INT nsp) VOID:
     IF curr_state /= str_st THEN
          change_state(space_in);
          IF outline /= brind THEN # there is text #
              output_string (outline) FI;
          breakpoint := nsp+1;
          brind := outline := indent := nsp*" " FI;
BOOL pragbody := FALSE;  #TRUE while outputing pment body #
STRING pragdel;  # save delimiter for end of pment #
PROC put_pragment = VOID:
     IF curr_state /= pment_st THEN
          change_state (pment_in);  pragbody := FALSE
     ELIF NOT pragbody THEN
          pragbody := TRUE;  pragdel := last_tok_out
     ELSE +/:= pragdel;  change_state(pment_in) FI;

PROC put_std = (INT stdno) VOID:  # one of the standard symbols #
     CASE std_type(stdno) IN
#1 monad #          put_monad(std_repr(stdno)),
#2 nomad #          put_nomad(stdno),
```

```
#3 :=,=:          put_asgn(stdno),
#4 space #        put_spaces(1),
#5 " #            (change_state(quot_in);
                       skipping_spaces := (curr_state = str_st);
                       +/:= """"),
#6 error #        (error("Unknown standard code.");
                       change_state(std_in);  +/:= std_repr(stdno)),
#7 other #        (change_state(std_in);  +/:= std_repr(stdno)),
#8 boldother #    (change_state(bold_in); +/:= std_repr(stdno);
                       tagstart := TRUE),
#9 boldmonad #    put_bold_monad(std_repr(stdno)),
#10 visible space # IF curr_state = str_st THEN +/:= " "
                  ELSE error("Visible space outside string.");
                       put_spaces(1)  FI  ESAC:  #end of put_std #
INT tao_cnt := 0;    # keeps track of how much tao has been put #
BOOL trans_tao := FALSE;  # TRUE if had to transliterate monad #
PROC put_monad = (STRING mon_repr) VOID:
     (change_state (tao_in);   tao_cnt := 1;
     trans_tao := FALSE;   +/:= mon_repr);
PROC put_bold_monad = (STRING mon_repr) VOID:
     (change_state(bold_in);   tao_cnt := 1;
     trans_tao := TRUE;   +/:= mon_repr);
PROC put_nomad = (INT nomad_num) VOID:
     (CASE tao_cnt + 1 IN
         # 0 # (change_state(tao_in);
                  tao_cnt := 1;   trans_tao := FALSE),
         # 1 # tao_cnt := 2,
```

```
        # 2 # error("TAO too long.")   ESAC;
    +/:= (NOT trans_tao ? std_repr(nomad_num)
        ? INT x;  char_in_string(REPR nomad_num, x, nomad_codes);
            "TMDVEQLTGTST"(2*x-1 : 2*x)));
PROC put_asgn = (INT asgn_num) VOID:
    (IF tao_cnt = 0 THEN change_state(tao_in);
            trans_tao := FALSE FI;
    +/:= IF tao_cnt > 0 & trans_tao THEN
            (asgn_num = 17 ? "TO" ? "AB")
        ELSE std_repr(asgn_num) FI;
    tao_cnt := 0);

(0:63) STRING std_repr;  # usual text for each standard code #
(0:63) INT std_type;     # for CASE in put_std #
BEGIN # this block is a documentary device.  It is used to
        introduce 't' so as to list in parallel the
        initial values for std_repr and std_type #
    INT monad = 1, nomad = 2, asgn = 3, space = 4,
        quote = 5, error = 6, other = 7, bold_other = 8,
        bold_monad = 9, visible_space = 10;
    (0:63) STRUCT (STRING std_repr, INT std_type) t =
        ((# 0 space   #     " ",        space),
        (# 1 stick   #     "?",        other),
        (# 2 quote   #     """",       quote),
        (# 3 hash    #     "#",        other),
        (# 4 dollar  #     "$",        other),
        (# 5 over    #     "%",        monad),
        (# 6 and     #     "&",        monad),
```

```
(# 7 q image #    """""""",      other),
(# 8 l paren #     "(",          other),
(# 9 r paren #     ")",          other),
(#10 times   #     "*",          nomad),
(#11 plus    #     "+",          nomad),
(#12 comma   #     ",",          other),
(#13 minus   #     "-",          monad),
(#14 point   #     ".",          other),
(#15 slash   #     "/",          nomad),
(#16 equal   #     "=",          nomad),
(#17 assigns #     "=:",         asgn),
(#18         #   " ERR18' ",     error),
(#19         #   " ERR19' ",     error),
(#20 flr(lwb)#    "LWB",         bold_other),
(#21 flr(ent)#   "ENTIER",       bold_other),
(#22 floor   #     "FL",         bold_monad),
(#23 ceiling #    "UPB",         bold_monad),
(#24         #   " ERR24' ",     error),
(#25 percent #     "PC",         bold_monad),
(#26 colon   #     ":",          other),
(#27 semicol #     ";",          other),
(#28 lessthan#     "<",          nomad),
(#29 becomes #     ":=",         asgn),
(#30 greater #     ">",          nomad),
(#31 cent    #     "CO",         bold_other),
(#32 at      #     "@",          other),
(#33 amper   #     "AM",         bold_monad),
```

```
(#34 window  #    "ELEM",      bold_monad),
(#35 up(shl) #     "SHL",      bold_other),
(#36 down    #     "SHR",      bold_monad),
(#37 subten  #      "e",       other),
(#38 of      #      "-<",      other),
(#39 grt eq  #      ">=",      monad),
(#40         #  " ERR40' ",    error),
(#41 plus i  #     "+*",       monad),
(#42         #  " ERR42' ",    error),
(#43 til(not)#     "NOT",      bold_other),
(#44 less eq #     "<=",       monad),
(#45 mod     #     "%*",       other),
(#46 not     #      "¬",       monad),
(#47 til(skp)#     "SKIP",     bold_other),
(#48 **      #     "**",       other),
(#49         #  " ERR49' ",    error),
(#50 up(pow) #     "**",       other),
(#51 asterisk#      "*",       nomad),
(#52 tilde   #     "TL",       bold_monad),
(#53 uparrow #     "UP",       bold_monad),
(#54 or      #     "OR",       bold_monad),
(#55 again   #     "?:",       other),
(#56 isnt    #    ":/=:",      other),
(#57 is      #     ":=:",      other),
(#58 differs #     "/=",       monad),
(#59 sub     #      "(",       other),
(#60 backsls #      "e",       other),
```

```
        (#61 bus     #      ")",        other),
        (#62 nil     #     "NIL",       bold_other),
        (#63 vis spc #      " ",        visible_space));
    std_repr := std_repr-<t;
    std_type := std_type-<t
END;  # block to initialize std_repr and std_type #
STRING nomad_codes = REPR 10 + REPR 15 + REPR 16 +
                     REPR 28 + REPR 30 + REPR 51;
          # times, divide, equal, less, greater, asterisk #

PROC is_reserved = (STRING tag) VOID:
    (INT reserved_limit = 7, # length of longest reserved word #
         number_reserved = 98,  # algorithm good to 128 #
IF UPB tag > reserved_limit THEN FALSE
ELSE (1:number_reserved) STRING reserved_word =
        ("ABS", "AND", "ARG", "AT", "BEGIN", "BITS", "BOOL", "BY",
        "BYTES", "CASE", "CHANNEL", "CHAR", "CO", "COMMENT",
        "COMPL", "CONJ", "DIVAB", "DO", "DOWN", "ELEM", "ELIF",
        "ELSE", "EMPTY", "END", "ENTIER", "EQ", "ESAC", "EXIT",
        "FALSE", "FI", "FILE", "FLEX", "FOR", "FORMAT", "FROM",
        "GE", "GO", "GOTO", "GT", "HEAP", #I# "IF", #IM# "IN",
        "INT", "IS", "ISNT", "LE", "LENG", "LEVEL", "LOC", "LONG",
        "LT", "LWB", "MINUSAB", "MOD", "MODAB", "MODE", "NE",
        "NIL", "NOT", "OD", "ODD", "OF", "OP", "OR", "OUSE", "OUT",
        "OVER", "OVERAB", "PAR", "PLUSAB", "PLUSTO", "POW", "PR",
        "PRAGMAT", "PRIO", "PROC", #RE# "REAL", "REF", "REPR",
        "ROUND", "SEMA", "SHL", "SHORT", "SHORTEN", "SHR", "SIGN",
        "SKIP", "STRING", "STRUCT", "THEN", "TIMESAB", "TO",
        "TRUE", "UNION", "UP", "UPB", "VOID", "WHILE");
```

```
    STRING uctag := tag;
    FOR i TO UPB uctag DO INT loc;
         IF char_in_string (uctag(i), loc, "abcdefghijklmnopqrstuvwxyz")
         THEN uctag (i) := "ABCDEFGHIJKLMNOPQRSTUVWXYZ" (loc) FI OD;
    INT loc := 64, incr := 32;  BOOL found := FALSE;
    TO 7 WHILE NOT found DO
         IF reserved_word(loc) > uctag THEN loc -:= incr
         ELIF reserved_word(loc) = uctag THEN found := TRUE
         ELSE loc +:= incr;
              IF loc > number_reserved THEN loc := number_reserved FI
         FI;   incr %:= 2  OD;
    found  FI); # end of is_reserved#

# input of codes  &  definition of codes (code 10) #

INT curr_code;  # filled by getch #

CHAR input_char;  # used by getch and logical_file_end #

FILE code_in;

     open (code_in); make_conv (code_in, complete_conv);

     on_logical_file_end (code_in, (REF FILE f) BOOL:

               (input_char := REPR 5;  # new page code #

               decoding := FALSE;    TRUE));

# definitions are stored in the following #

(0:63) FLEX (1:0) INT definition;  # store definition #

FLEX (1:0) INT inputs;  # copy definition for reading by getch #

INT inptr := -1;  # pointer into inputs used by getch #

PROC set_define = VOID:    # called in response to code 10 #

     IF INT code = getch;  INT len = getch;  # not collateral #

               len = 0  THEN  # restore prescribed meaning #

          code_function (code) := perm_code_function(code);

          definition(code) := ()

     ELSE # define the code to be the next 'len' codes #
```

```
        code_function (code) := put_defn;
        definition (code) := HEAP (1:len) INT;
        FOR i TO len DO
             definition (code)(i) := getch OD FI;

PROC put_defn = VOID:    # code_function (defined code) = put_defn #
      (INT ui = UPB inputs - inptr + 1,
          ud = UPB definition (curr_code);
      (1 : ui+ud) INT tin;
      tin(1:ud) := definition (curr_code)(:);
      tin(ud+1:) := inputs (inptr:);
      inputs := tin;   inptr := 0);
PROC getch = INT:    # called on to get a code from input #
      IF inptr < 0 THEN get (code_in, input_char);
          curr_code := ABS input_char
      ELIF inptr >= UPB inputs THEN inptr := -1;   getch
      ELSE curr_code := inputs(inptr+:=1)  FI;

# output routines and routines to access output string:  outline #
STRING outline := "",    # line being built #
     indent := "",       # blanks put at start of line by new line #
     brind := "",        # blanks put at start of most recent line
                                 (as possibly broken) #
INT mxln = 72,    # maximum number of characters in line #
      breakpoint := 1,    # start of most recent token
                                 (a potential line break spot) #
      linenumber := 0;    # for numbering output lines #
FILE ra68_out;   open (ra68_out);   # the reconstituted text #
OP +/:= = (STRING s) VOID:   outline +:= s;
```

```
PROC line_has_left = (INT n) BOOL:  mxln - UPB outline >= n;
PROC last_tok_out  = STRING:  outline (breakpoint:);
PROC set_break = VOID:
      (WHILE UPB outline > mxln DO break_line OD;
      breakpoint := UPB outline + 1);
PROC output_string = (STRING line) VOID:
      (STRING tline = line + (mxln - UPB line + 1)*" "
               + whole (linenumber +:= 100, 7);
      put (ra68_out, tline);
      put (standout, tline));
PROC break_line = VOID:    # terminate input line longer than mxln #
    IF breakpoint = UPB brind + 1  &  brind /= ""
               & curr_state = bold_st THEN
          # no breakpoint, unindent line #
          outline := last_tok_out;  breakpoint := 1
      ELSE # move last token to new line rather than split it #
          brind := # indent for rest of line #
                   (UBP indent > mxln%2 + 5
                        ? (mxln%2)*" " ? indent+7*" ");
          IF breakpoint = 1 THEN # must split token anyway #
              breakpoint := mxln+1;
              IF curr_state = bold_st THEN    # truncate bold #
                   outline := outline(1:mxln)
              ELIF curr_state = str_st THEN brind := ""
              ELIF curr_state = tag_st THEN # insert "_" #
                   outline := outline(1:mxln) + "_"
                        + outline(mxln+1:)   FI  FI;
```

```
            IF curr_state = out_st THEN skipping_spaces := TRUE FI;
            output_string(outline(1:mxln));
            outline := brind + last_tok_out;
            breakpoint := UPB brind + 1
        FI;  # end break_line #

# errors #
INT nerrs := 0;
PROC error = (STRING msg) VOID:
        nerrs +:= 1;
        # in an implementation, the "msg" would be printed
            with a pointer to its location #

# main program #
BOOL decoding := TRUE;     # set FALSE by logical_file_end(code_in) #
WHILE decoding DO
        code_function(getch)    OD;
put (standout, "There were "+whole(nerrs,0)+" errors.")
END;# of decoder #
```

IBLIOGRAPHIC DATA HEET	1. Report No. UIUCDCS-R-74-622	2.	3. Recipient's Accession No.
Title and Subtitle Transportation of Higher-Level Language Programs: Exemplified by an ALGOL 68 Transportation Representation			5. Report Date January, 1974 6.
Author(s) Wilfred J. Hansen			8. Performing Organization Rept. No.
Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
. Sponsoring Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801			13. Type of Report & Period Covered 14.

. Supplementary Notes

. Abstracts

Problem: Modern higher level languages, for example, ALGOL 68, must be defined to allow many alternate token representations to accommodate the variety of input/output equipment available. Because of this variety, it may not be possible to transport programs from one hardware representation to another by any simple process of substitution.

Solution: As an example of the proposed approach, this paper defines a six-bit encoding for ALGOL 68 program texts. It is an encoded character prefix representation, so it can be decoded without lookahead. The encoding is defined in detail and a decoder for a specific hardware representation is given.

Key Words and Document Analysis. 17a. Descriptors

. Identifiers/Open-Ended Terms

. COSATI Field/Group

Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page UNCLASSIFIED	22. Price

MAY 5 1977

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.619-626(1974
Guide an information system.
3 0112 088401143